Access-Aware DRAM Failure-Rate Estimation under Relaxed Refresh Operations

Access-Aware DRAM Failure-Rate Estimation under Relaxed Refresh Operations

Konstantinos Tovletoglou, Dimitrios S. Nikolopoulos, Georgios Karakonstantis
Electronics, Electrical Engineering and Computer Science, Queen's University Belfast
Email: {ktovletoglou01, d.nikolopoulos, g.karakonstantis}@qub.ac.uk

***Abstract*—In recent years, there has been a growing interest on relaxing the pessimistic DRAM refresh rate due to the incurred power and throughput loss. Undeniably, a critical factor in determining the refresh rate relaxation that can be achieved lies on the degree of the DRAM error-rate deterioration that is incurred and on the amount of estimated errors that can be handled by system mitigation schemes which are mainly being evaluated in simulators. To estimate the DRAM faults under relaxed refresh, the majority of the existing works rely on estimated DRAM failure probability models using only the spatial distribution of the DRAM retention time across the memory cells. We observe that such failure models have neglected the intricate dependence on the memory accesses, which inherently refresh the accessed rows.**

In this paper, we propose that the intervals between consecutive accesses must also be considered during DRAM simulation. We show that the estimation of the distribution of accesses poses a lot of challenges mainly due to the time consuming full system simulations that are required. To address such challenges, this paper presents one of the first efforts to model the access time-dependent DRAM retention time by developing a fast simulation infrastructure based on binary instrumentation. The basic idea behind the proposed approach lies on the quantification of the time elapsed between consecutive memory accesses on the same row and its relation to the DRAM failure probability, which is then being used for a more accurate fault injection. The introduced overheads of the instrumentation functions are measured during native execution allowing accurate corrections of the time elapsed between consecutive accesses. The efficacy of our framework is being evaluated using various artificial benchmarks. Results show that our scheme helps to limit the misprediction of estimated errors of current error-injection models.

***Keywords*: memory tracing; fault injection; DRAM memory; retention time; simulation; binary instrumentation;**

I. Introduction

Challenges such as Dark Silicon [1], according to which only 79% of the silicon at 22 nm may be powered at any point of time due to the increased power density in multicore systems, have raised power as a primary concern. Such a challenge critically affects the performance scaling of servers which are essential for supporting the emerging Cloud computing paradigm. In the past decade, power consumption of server systems was attributed mainly to CPUs, but recent studies [2], [3], [4] turned the attention also to the memory hierarchy, which has started being responsible for large percentages of the power consumption. Traditionally Dynamic Random-Access Memory (DRAM) is used as main memory due to the low cost and high density storage that it offers. As the data generated by all the connected digital devices continue to rise the demand for higher DRAM densities will become larger and thus the power consumed by the DRAMs will keep rising [5].

The scaling of DRAM is partially hindered by the periodic refresh cycles that are needed to compensate for the leaking charge of the DRAM cells [6]. Conventionally, the refresh period (T_{REFW}) for the whole DRAM array is determined by the worst case retention time of the most leaky cell. Such an approach incurs large power and throughput overhead, which is expected to be around 40% and 30% respectively in future DRAM generations [7]. To circumvent such overheads, recent approaches have experimented with relaxed refresh rates by utilizing the spatially varying retention time of the DRAM cells [7], [8] or temporal characteristics of the DRAM, such as [9], [10], [11] while ensuring fault-free operation. Works, such as [12], [13], [14], try to exploit the error-resiliency of various applications for tolerating any DRAM errors under aggressive refresh-rate relaxation. Common evaluation platform for the majority of the existing schemes are system simulators mainly due to the practical limitations of the available memory controllers.

While the used DRAM simulators [15], [16] are widely acceptable, none of the existing fault-injection schemes [17], [18] is taking into account the refresh by access. Such schemes have focused only on the spatial variation of the retention time of each cell within the memory array, based on extracted failure rates over different refresh periods.

Our experiments have indicated that various popular applications access the memory frequently and thus naturally refresh the stored data, a property which should be taken into account while estimating the DRAM failure rate. However, modelling such a property and considering it during simulation poses a series of challenges.

One may try to extract the time elapsed between consecutive access to a specific row but this requires full system simulations, within for instance simulators such as gem5 [19], which are extremely time consuming. Coping with the huge overhead required for keeping track of the elapsed time between memory accesses for each individual DRAM row, may be prohibitive especially for large memories and long running applications.

In this paper, we aim at addressing the above challenges by developing an infrastructure for quick and accurate estimation of the intervals between DRAM accesses which can then be used for a more precise fault injection than existing schemes.

Our contributions can be summarized as follows:

- We develop a methodology based on a series of tools that extract the elapsed time between consecutive memory

accesses, while executing a benchmark natively through binary instrumentation. Our approach help us to avoid the extremely slow full system simulations that otherwise may be required.
- We show case the usage of the method by implementing a fast fault-injection simulation infrastructure which utilizes the extracted elapsed timings between accesses for estimating how many errors will be injected and compare them to existing schemes. The proposed approach allows the evaluation of the impact of the potential errors incurred under relaxed DRAM refresh operations on any application by considering the implicit refresh by access.
- Finally, we evaluate the accuracy of our framework by using artificial benchmarks, that are composed of a various access patterns and workload sizes inspired by real benchmarks. Our results indicate that the accuracy of our scheme is high for larger workloads and regular access patterns but it degrades as the complexity of the benchmark, and thus irregularity increases.

The rest of the paper is organized as follows. Section II describes some basic background on DRAM operation. Section III briefly presents recent related work. Section IV describes the proposed approach for extracting timing traces of accesses to the memory. In Section V, we present a case study of our approach for estimating the error probabilities under relaxed refresh operation. In Section VI, we discuss the evaluation of our approach, presenting the artificial benchmarks and the accuracy of our scheme while being compared to existing schemes. Finally, conclusions are drawn in Section VII.

II. Background

In this section, we briefly describe the DRAM organization and the refresh operation.

A. DRAM organization

A main memory system based on DRAMs is organized hierarchically into channels, ranks, banks, rows and columns, as shown in Figure 1. Each DRAM module (referred to as DIMM) usually has two ranks consisting of a number of two-dimensional arrays of DRAM cells, the so called banks. Each DRAM cell is a storing element of DRAM and consists of a capacitor and an access transistor. Each access transistor connects its associated capacitor to a wire called a bitline and is controlled by a wire called wordline. Cells sharing a wordline form a row. Before a row can be activated, all bitlines in the bank must be precharged. The row's wordline is enabled by connecting all capacitors in that row to their respective bitlines. This causes charge to flow from the capacitor to the bitline. Finally, the sense amplifier connected to that bitline detects the voltage change and amplifies it, driving the bitline fully either to the power rail or to zero voltage.

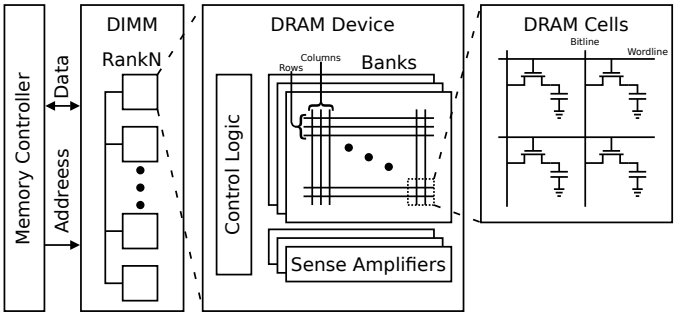


Fig. 1: DRAM memory system organization

B. Retention Time and Refresh Operation

The simple structure of the DRAM array and of each DRAM cell may allow high storage density, however is not capable of retaining the stored charge for a long period due to the inherent transistor's leakage current. Such leakage can eventually discharge the cell, manifesting a bit-flip. The duration that the cell can correctly retain its state (i.e. '0' or '1') without eventually experiencing any bit-flip is called *retention time*.

To avoid any error induced by the limited retention time, modern day DRAMs employ an *Auto-Refresh* mechanism that periodically recharges each cell in the DIMM by simply bringing the data from a row into the sense amplifiers and restoring them back in the row. To achieve this, the memory controller issues a refresh command every t_{REFI} cycles, at which point all DRAM banks simultaneously refresh a number of rows making the rank unavailable for t_{RFC} cycles.

Currently, the refresh period T_{REFW}, i.e. the interval within which all cells of the DIMM must be refreshed, is set according to the worst case retention time of all cells. In fact all DDRx technologies adopt today a T_{REFW} of 64 ms under nominal environmental conditions or 32 ms in case of temperatures higher than 85°C.

Such a refresh period leads to considerable power and performance overheads, which are expected to worsen as the DRAM density increases [20].

Finally in current DRRx technologies, the refresh operation and memory accesses are mutually exclusive to each other. This means that no row/chip within a rank are allowed to be accessed while the specific rank is being refreshed. This does not only cause a performance penalty, but also prevents the exploitation of the implicit refresh incurred with each memory read.

III. State-of-the-Art and Limitations

A. Schemes for Error-Free DRAM Operation

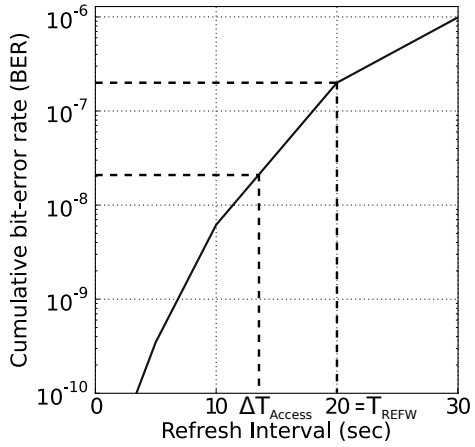
In an attempt to address the refresh related overheads, recent studies have shown that the retention time of cells varies considerably across and within a DRAM chip. Typically, only a very small number of cells needs to be refreshed once every $T_{REFW} = 64\ ms$ [21], [22], [20], [23]. Techniques proposed in works such as [7], [8], [24], [25], [26], [27], [12], try to exploit the spatial non-uniformity in retention time of DRAM cells to reduce the frequency of DRAM refresh.

The work in [9] was the first to exploit the temporal characteristics and the implicit refreshes through DRAM accesses by keeping a log of the recent accesses to the memory controller in order to skip refresh operations to specific rows. Similar works are described in [10], [11], which are focusing in relaxing the refresh rate on eDRAM caches.

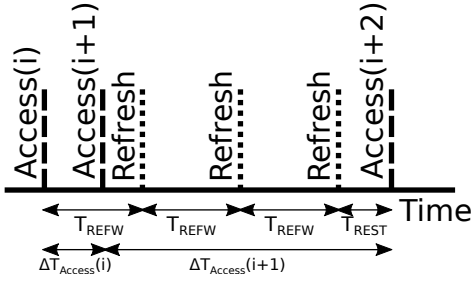
All the above works may have tried to reduce the refresh, while avoiding any fault but such approaches are highly intrusive since they assume fine grain control of the refresh rate and/or require monitoring of every single access to every row, which require substantial changes in the DRAM controller. This makes such schemes difficult to realize in practice since controllers are currently providing restricted access even to basic parameters (e.g. refresh rate). Due to the intrusive required modifications such schemes have been mainly evaluated on simulators by relying on extracted failure rate models, which have neglected the implicit refresh by access, which may lead to over- or under-estimation of the refresh rate relaxation that can be achieved.

B. Error Resilient Schemes

Another common feature of the works mentioned above is that they still try to avoid any DRAM faults neglecting the error resiliency of various applications, which may limit the achieved gains. In this context, some recent works have tried to further relax the refresh rate of the previous conservative methods by extending refresh intervals to regions that errors occur and they propose solutions to address the resulting errors either by correcting them through error-correction codes [12] or exploiting the paradigm of approximate computing and error-resiliency of the application for reducing their impact on quality [13], [14].



(a) Cumulative distribution function of bit-error with relaxed refresh rate or access interval. Visible is the decreased probability of error when the access interval is smaller than the refresh interval.



(b) Timing of refreshes inbetween accesses, showing the case of underestimating the probability for error injection of state-of-the-art simulation schemes.

Fig. 2: Fixed probability fault-injector deficiencies.

Such studies are being evaluated on simulators by relying on fault-injection schemes under a selected refresh-rate based on a fixed DRAM failure probability which ignores the temporal characteristics of DRAM and the implicit refresh that takes place during every access. Figure 2a shows a cumulative density function (CDF) of the bit-error rate over different refresh intervals which is being extracted by ignoring the implicit refreshes. CDFs similar to the one shown in Figure 2a are being being used in the majority of the above studies for injecting faults [14] according to a fixed failure probability of the selected refresh interval, e.g. for a T_{REFW}=20 sec.

C. Limitations in Failure Probability Estimation

However, if we also consider the refresh by access then there are two scenarios in which the conventionally estimated failure probability will be under- or over-estimated.

On the one hand, if the same row is accessed within a time less than the selected T_{REFW} then the estimated failure probability will be less than the one being estimated if we consider only the failures at a fixed T_{REFW}. In this case the failure probability is over-estimated by neglecting the memory accesses. On the other hand, if the same row is not accessed for a time longer than the selected refresh interval, then there is also possibility for an error to occur during the period between two refresh operations. Such a case was also neglected by recent schemes and assumed no errors leading to an underestimated failure probability.

To better understand the two scenarios let us consider an example as depicted in Figure 2b and assume that a running application triggers Nr memory accesses to the r row in memory so that the intervals between consecutive accesses to the same row, $\Delta T_{i,r}$, can be calculated as:

$$\Delta T_{i,r} = T_{Access_{i+1,r}} - T_{Access_{i,r}} \quad (1)$$

As shown in Figure 2b, for the case of $T_{Access}(i)$ and $T_{Access}(i+1)$, the $\Delta T_{Access}(i)$ is less than the T_{REFW}. In this case, the failure probability should be calculated based on the CDF at ΔT and eventually will be smaller than the fixed one based on T_{REFW} and thus may be overestimated . Furthermore, for the case of $T_{Access}(i+1)$ and $T_{Access}(i+2)$, the $\Delta T_{Access}(i+1)$ is longer than the T_{REFW}, as shown in 2b then each time the row is refreshed there is probability that error may be introduced based on the selected T_{REFW}. However, in the existing schemes the failure probability is only calculated once under a selected and fixed T_{REFW} which may cause under-estimation.

Based on the above, it is evident that for estimating accurately the DRAM failure probability, there is a need to also consider the intervals between consecutive accesses to each row, as aimed by this paper.

To achieve this there is a need essentially to extract the timing trace of the memory accesses to each row. An accurate method to achieve this is by using a full system cycle accurate simulator, such as gem5 [19]. However, full system simulation is extremely time consuming and essentially prohibitive for long running programs. In particular, to quantify the time

needed in obtaining the timing trace from a full system simulator we have executed on gem5 and it is around 20.000 times slower than native execution as it will be shown later in the Section VI.

D. Overcoming the Limitations

The above overheads could be limited by utilizing other memory tracing schemes which involve memory access sampling and which were used for performance estimation and optimization [28], [29]. However, utilizing such schemes for calculating the intervals between accesses of all the rows may not be appropriate, since there is a need of accurate estimates of every access. Failing to capture even one access may result in over estimating the intervals and as a consequence lead to an erroneous fault injection.

An alternative scheme for tracing the memory accesses and estimating the intervals may be based on binary instrumentation. A variety of frameworks exist such as Intel® Pin [30], DynamoRIO [31] and Valgrind [32], which provide instruction-level and code-block instrumentation of the binary code. Binary instrumentation allows to inject code on selected instructions, i.e. all instructions that access the memory. Instrumentation can take measurements during the execution from simple counters up to complex systems, i.e. simulating cache hierarchies. Through instrumentation, it is allowed to record the times of each access using the X86 assembly instruction $rdtscp$ [33], which can provide accurate measurements on serialized code. However, the instrumentation affects the measurements as it injects more code for each invocation. Therefore, it needs to be carefully embedded and considered within any measurement campaign.

IV. Proposed Approach

The primary aim of this paper is the development of a framework that allows the estimation of DRAM failure rate under relaxed refresh rate by considering the implicit refresh by access property. The main idea of our approach lies on measuring the intervals between consecutive accesses to each memory row and using them for estimating the DRAM failure rate.

The intervals between consecutive accesses incurred by a target application are traced natively with binary instrumentation. The estimated failure rate can then be used for injecting faults during functional simulation of any target benchmark for evaluating the impact of potential errors.

A. Dynamic Binary Instrumentation

For the development of the proposed framework we used Intel®'s Pin [30] dynamic binary instrumentation tool. Pin allows us to analyse the instructions of the application and inject a branch and an instrumentation function for collecting the intervals between consecutive memory accesses.

As we mentioned, any dynamic binary instrumentation function adds overheads in the execution time that need to be carefully considered during the experimental campaign. The largest part of the overhead is introduced by the execution of the injected routine used for extracting the timing traces and it clearly depends on the required instructions. One of our ideas is to estimate such an overhead during each run and subtract it from the total execution at the end. A part of such an overhead is introduced by the call and exit of the branch and cannot be measured directly. However, since it is constant across different invocations, we can estimate an adequate average during the execution.

In addition, the instrumentation function inflict corruption of the cache of the running benchmark. In order to keep the corruption of the cache limited, we are trying to minimize the data-structure size of the instrumentation code. Also each time that the instrumentation routine is called, we are executing the same flow of code so that it inflict the same cache misses, eventually flushing a part of the cache each time.

B. Phases of the Proposed Framework

Our framework consists of distinct phases during which the timing tracing of the memory accesses along with the estimations of the introduced overheads during the execution of a specific benchmark take place. The block diagram of the different phases are shown in Figure 3 and each phase is described in the following paragraphs.

1) Native Run: Initially, we execute the application natively on the machine to extract the total execution time, T_{Native}, without any modification and instrumentation. This helps to establish the expected execution time of the target benchmark and use it as reference for calculating the overheads that are introduced by the instrumentation.

2) Calculating Static Overheads: Afterwords, we execute the application on top of the Pin framework loading all the instruments that we will use later but without invoking them in the execution. In this way, instrumentation is limited only on the loading the Pin framework and on the initialization of our tools at the start of the application avoiding to introduce any overhead for each instruction. We measure the total execution time, T_{Tools}, and we calculate the static overhead that is introduced by the Pin framework during the start of the application.

3) Timing Trace of Accesses: Subsequently, we execute the application, loading our instrumentation and enabling the invocation of the instrumentation function. We measure the total execution time, T_{Full}, the times that our instrumentation was invoked, $TotalInvokesCount$, and we keep in a trace a list of information vectors (IV) with the intervals of consecutive access to the each row, ΔT_{Access}, and the number of invocations of the instrumentation since the previous access, $InvokesCount$.

Each time we invoke our instrumentation, we update the accumulated overhead of our instrumentation, $T_{Current}^{Overhead}$, based

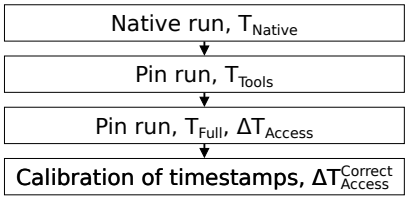


Fig. 3: Block diagram showing the phases of our instrumentation

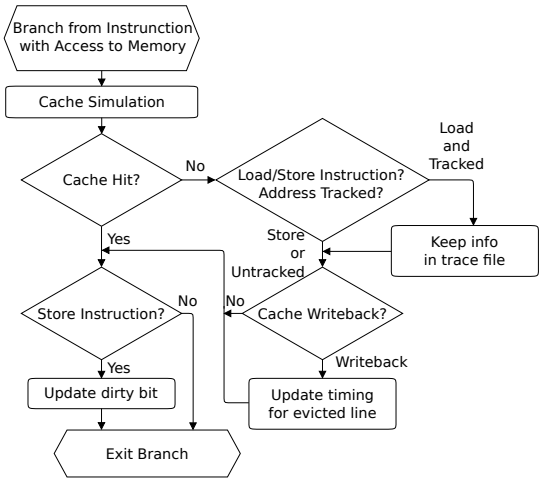


Fig. 4: Data flow of the instrumentation function called for each instruction that accesses memory

on the initial timing measurement after the branch, entering the instrumentation code, T^{Branch}_{Start}, and before exiting the instrumentation, T^{Branch}_{Exit}. Even though the timing measurements of the instrumentation function is as close as possible to real ones, there are overheads that cannot be tracked down, $T^{Overhead}_{Unknown}$, that we consider as zero in this step and will correct next. So the accumulated overhead can be calculated

$$T^{Overhead}_{Current} = T^{Overhead}_{Current} + T^{Branch}_{Exit} - T^{Branch}_{Start} + T^{Overhead}_{Unknown}$$

We calculate the ΔT_{Access} taking into account and subtracting the accumulated overhead since the previous access to this row, $T^{Overhead}_{LastAccess}$.

$$T^{Overhead}_{LastAccess} = T^{Overhead}_{Current} - T^{Overhead}_{Previous}$$

$$\Delta T_{Access} = T^{Current}_{Access} - T^{Previous}_{Access} - T^{Overhead}_{LastAccess}$$

The logical flow of our Pin instrumentation and decisions about which accesses will be stored in the trace file is shown in Figure 4 and is described next. Our instrumentation function is invoked for all instructions that are accessing the memory and injects a branch instruction to the instrumentation function. We keep the IV of accesses in the trace file when a load instruction causes a cache miss and the given address of the data is inside the region of the application that we keep track. The IV of an access consists of the address, the count of instrumentation calls and the timing since the last access and the total so far. If the cache has a conflict based on the replacement scheme of the memory, we evict a cache line and if needed, the data are written back to memory and we have to update the last access time of the evicted line. In the end, we check if it is a store instruction, in order to update the dirty bit of the given address.

4) Calibration of Measurements: As described above, in the previous step we introduced $T^{Overhead}_{Unknown}$ which could not be estimated during the tracing. Since this overhead is caused by not measuring the time spent on the branch instruction itself, the exit of our routine and the breaking of the cache, we consider it to be static across each invocation of our instrumentation function. So the information from the previous steps is used to estimate it. In particular, we calculate an average value of this overhead for each invocation by using the following formula:

$$T^{Overhead}_{PerAccess} = \frac{T_{Full} - T_{Native} - T_{Tools} - T^{Overhead}_{Total}}{InvokesCount_{Total}}$$

At the end, we recalibrate the measurements in the trace file to have accurate timings, according the following formula:

$$\Delta T^{Correct}_{Access} = \Delta T_{Access} - InvokesCount * T^{Overhead}_{PerAccess}$$

V. Case Study: Failure-Rate Estimation under Relaxed Refresh Operation

As we discussed the primary use of the extracted timing traces is on improving the existing DRAM fault injection schemes. In this section, we describe as a case study how our framework can be used for estimating and injecting faults under relaxed refresh operations during application execution.

A. Error Generation

Following the measurement of the intervals between consecutive memory accesses occurring within the target application, as described in the previous section, we then estimate the memory faults that need to be introduced. In particular, we calculate the probability of an error for each access that goes to the memory controller based i) on the extracted intervals between consecutive accesses of the same row and ii) the Cumulative Distribution Function (CDF) of bit-error probability from 1 up to 150 seconds of T_{REFW} Note that in our case, we have extracted the CDF from a server system the details of which are described on Table 1. Our experiments were performed on the server's DRAMs consisting of 8 DIMMs of 8 GB DDR3 at 1600 MHz. By disabling the refresh and using test kernels that write specific patterns [34] in memory and access them after a specific amount of time we characterized the T_{REFW} of each cell similarly to [20]. The extracted CDF for each DIMM for a range of T_{REFW} from 1 up to 150 seconds is shown on Figure 5.

Using the extracted CDF and the measured intervals between accesses, we can estimate the fault injection probability in any specific row. In particular, if the access interval is greater than the T_{REFW} then we check the probability multiple times, specifically N times as calculated by the following

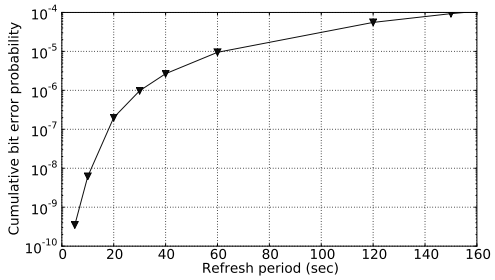


Fig. 5: Cumulative bit error probability for a 64-bit word with variable refresh interval

function, using the error probability of T_{REFW} and for the last time, we fault-inject based on the probability of the remaining time, T_{Rest} from the function. We create a new trace file only with the IV that will be used to fault-inject by the next step.

$$\Delta T_{Access}^{Correct} = X * T_{REF} + T_{Rest}, \qquad T_{Rest} < T_{REF}$$

This phase allows us to estimate quickly the injected errors using one trace of the IVs. By using inputs from a randomization seed we then generate a number of error traces that can be injected and used for evaluating the application behaviour.

B. Fault Injection

Having estimated the faults that need to be introduced, we can then execute the application while performing fault-injection within a functional simulator, which is built around also Pin. The instrumentation function for this step is called by branching the execution of the application on instructions with accesses to the memory. Only the rows specified by the error trace of the IVs are going to be fault-injected and to identify which rows, we are using the address of the access and the number of accesses that happened so far.

During the execution, we monitor the output result evaluating if the application crashed. In case that the application is completed, we can estimate the output quality using application relevant error metrics.

VI. Evaluation Details and Results

To evaluate the efficacy of our framework we have developed artificial benchmarks with various access patterns and used an experimental setup based on a real server that could allow us to compare the estimated timing intervals.

A. Experimental Setup

The validation of the simulator is done using a real system, specifically a commodity server machine with dual-socket Intel processor. The specifications of the system are found on Table 1.

B. Benchmarks

The constructed artificial benchmarks resemble kernels with access patterns representative of real applications. Operations are being replaced with boolean operators that can help to trace back how many errors were introduced, while additional delays have been added for emulating the time spent on the processor.

Parameter	Value
CPU	Intel® Xeon® CPU E5-2650
Clock	2.00 GHz
Cores/socket	8
Sockets	2
L1 data cache/core	32 KB, 8-way associative
L2 cache/core	256 KB, 8-way associative
L3 cache/socket	20 MB, 20-way associative
Replacement scheme	Least Recently Used(LRU)
DRAM Type	DDR3
Memory Size	8 * 8 GB
Memory Clock	1600 MHz

Table 1: Specification of the hardware

The constructed artificial benchmarks vary in terms of the data-structure size, the type of the access patterns (i.e. sequential, random-walker, hybrid) and the frequency (rate) of memory accesses. Such artificial benchmarks allow us essentially to control the access patterns and thus evaluate the efficacy of our framework with different intervals between memory accesses and workload sizes.

C. Evaluation Flow of Timing Intervals

In the artificial benchmarks, we have instrumented native sampling of the time to validate the timing measurements of Pin. The sampling is done by accessing a unique memory address and keeping the timestamp of the access. By doing so, we can measure the intervals of the specific memory address without any Pin instrumentation. The introduced sampling is very sparse to avoid introducing any major overhead in the execution.

Afterwords, we execute the same benchmark through our framework to extract the memory trace and we correct the timestamps using the approaches discussed in Section IV. The intervals iobtained by the native execution of the specific memory address can be associated with accesses of the same address in the trace extracted from the Pin. We compare the measured intervals from the native instrumentation with the ones estimated by our framework. In this way, we can evaluate the accuracy of our timing trace estimation framework.

1) Accuracy of tracing: We measure the accuracy of our framework, based on the evaluation flow described in the previous section. Overall, we observe that the estimation accuracy of the proposed framework varies from 1.8 up to 17 times for benchmarks with small memory usage. However, the accuracy is improved for larger workloads.

In particular, by using the artificial benchmarks with the various access patterns but with memory up-to 512MB we observed that in worst case, we overestimate the timing intervals by up to 1.8 times compared to the intervals extracted by the native execution. There are also cases that our framework underestimates the timing intervals by up to 17 times when compared to the intervals extracted by the native execution.

Note that we have not experimented with benchmarks with less memory than the cache size as it is impossible to conclude anything meaningful from those experiments.

On the other hand, when we execute the same benchmarks but with datasets ranging from 512 MB up to 8 GB we observed that the accuracy of our framework is very close to the native execution. For benchmarks with only sequential accesses, that have similar workload across each access, our framework may overestimate the timing intervals from 1.09 up to 1.16 times. For benchmarks with only sequential accesses, that have heavily variable workload across each access, the intervals can be either be underestimated by up to 1.36 times or be overestimated up to 1.06 times. For benchmarks with only random accesses, that are instrumented in such way that every access is a cache misses, our framework may underestimate the timing intervals by up to 1.38 times and may overestimate them by up to 1.24. Similarly, the accuracy further decreases for benchmarks with only random accesses,

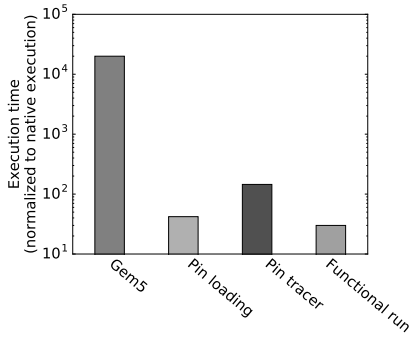


Fig. 6: Execution time of different techniques of timing trace and simulation, normalized to the execution time of the native execution of the benchmark

that may include cache hits, underestimating by up to 1.88 times and overestimating by up to 1.26.

The above results indicate that the accuracy of our scheme is high for larger workloads and regular access patterns but it degrades as the complexity of the benchmark, and thus irregularity increases. Furthermore, in cases that we the cache accesses have a pattern and are either most of them cache hits, such as the first case, or mostly cache misses, such as the second case, our framework tends to have better accuracy, compared to benchmark that have irregularities. Finally, we observe that our estimates of intervals on average may be under estimated, meaning by that we are potentially over-correcting the instrumentation overheads.

2) Overhead: Apart from the accuracy we also measure the overhead of our framework in terms of execution time. Figure 6 depicts the overhead introduced for each technique normalized to the execution time on the native system. Our framework has two phases for extracting the timing trace through Pin; one measuring the overhead for loading Pin and one for extracting the timing traces. These two phases are compared with the execution time in a full system cycle accurate simulator i.e. gem5.

The execution time of the phase that is just loading the tools is around 45 times slower than the native execution. We observed that the execution time of the benchmark can increase by up to 90 times in case that the benchmark is running for less than few seconds. In our experiments, the phase for extracting the trace can be up to 220 times slower than the native execution in the worst case of benchmarks with high cache miss rate, but on average it is 145 times slower than the native execution. As shown in Figure 6, gem5 is around 20.000 times slower than the native execution. In any case, our framework is orders of magnitudes faster when compared to gem5. The execution time of the functional simulator that is used for injecting faults in the benchmark according to the estimated errors is around 30 times slower than the native run.

Finally, Figure 6 shows the execution time of the error generation and of the functional run combined. The execution time of the functional simulator that is used for injecting faults in the benchmark according to the estimated errors is around 30 times slower than the native run, which is can be considered quick enough.

In the next section, we are presenting the results of our framework on the case study of fault-injection.

D. Evaluation of Fault-Injection

We are evaluating our fault-injection scheme by comparing it to a state of the art scheme.

1) Baseline Scheme for Fault-Injection: In terms of the fault injection scheme, we compare our approach with a scheme that injects DRAM faults according to a fixed probability of error, extracted by the CDF, each time the memory is read and the data do not reside in the cache. Specifically, as the fault injection baseline we are using the implementation described in [14]. The two schemes are evaluated by using different benchmarks based on the number of errors that are injected on the data structures.

2) Results: As we discussed in previous section, the extracted timing can be used to fault inject within a simulator following the steps discussed in Section V. Figure 7 shows the difference in the number of error occurred of the two fault-injection schemes, one with fixed probability as in [14] and our implementation taking into account the intervals of consecutive accesses. The benchmark is allocating 1 GB of memory and accessing all the rows every $\Delta T = 5sec.$. So even if we increase the T_{REFW} further than the ΔT, the data are refreshed by the accesses. Our proposed method results are the expected one, as we increase the T_{REFW} further than the ΔT, the probability remains stable, the one based on the intervals. For the scheme with fixed error probability for each access, the errors increase while we are increasing the T_{REFW} indefinitely, overestimating the errors. Furthermore, we can see that for $T_{REFW} = 2sec.$, the scheme with fixed probability is underestimating the errors, because it only check once for the probability based on 2 sec., while our scheme calculate the probability taking into account multiple cycles of T_{REFW} as described in Section V.

VII. Conclusions

This paper presents an access-aware DRAM failure rate estimation framework which can help improve the state of the art. As we discussed every memory access implicitly refreshes the DRAM rows and by neglecting such a property

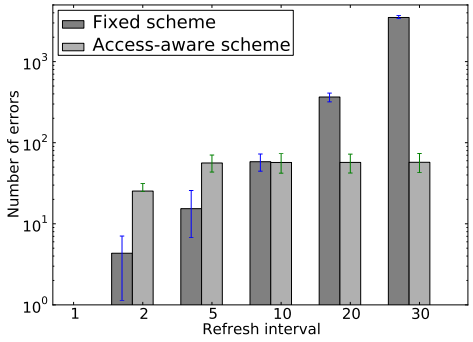


Fig. 7: Comparison of the fault-injection scheme with fixed probability of error and our access-aware scheme.

during fault injection then an over-estimation of the introduced errors may be incurred, which can lead to an under-estimation of the possible refresh-rate gains. Based on dynamic binary instrumentation the proposed framework allows us to extract the timing intervals between consecutive accesses for a target application quickly and accurately and use them for property injecting faults during functional simulation. Our framework outperforms other conventional methods based on cycle accurate simulators which can also be used for extracting the timing traces by 100 times. The proposed scheme provides adequate timing trace estimation accuracy, especially for workloads with a lot of memory and regular access patterns, when compared to the native execution of various benchmarks. Finally, it can help limit the over- or under- estimation of injected faults occurred by existing schemes.

ACKNOWLEDGEMENTS

The presented research effort was partially supported by the European Community's Horizon 2020 UniServer project (grant no. 688540) and FET-Proactive OPRECOMP project (grant no. 732631).

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39, 2011, pp. 365–376.

[2] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, Dec 2003.

[3] J. Meza, M. A. Shah, P. Ranganathan, M. Fitzner, and J. Veazey, "Tracking the power in an enterprise decision support system," in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 261–266.

[4] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, 2010, pp. 189–194.

[5] M. E. Tolentino, J. Turner, and K. W. Cameron, "Memory miser: Improving main memory energy efficiency in servers," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 336–350, March 2009.

[6] T. Hamamoto, S. Sugiura, and S. Sawada, "On the retention time distribution of dynamic random access memory (dram)," *IEEE Transactions on Electron Devices*, vol. 45, no. 6, pp. 1300–1309, Jun 1998.

[7] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent dram refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[8] R. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* IEEE, 2006, pp. 157–167.

[9] M. Ghosh and H. H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 134–145.

[10] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas, "Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies," in *19th IEEE International Symposium on HPCA '13*, pp. 400–411.

[11] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "Chargecache: Reducing dram latency by exploiting row access locality," *IEEE Symposium on HPCA '16*, pp. 581–593.

[12] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 72–83.

[13] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate DRAM?" in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, oct 2015, pp. 89–98.

[14] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 213–224.

[15] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan 2016.

[16] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[17] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr 1997.

[18] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[20] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, O. Mutlu, J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern DRAM devices," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*.

[21] I. Bhati, M. T. Chang, Z. Chishti, S. L. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 108–121, jan 2016.

[22] K. Kinam Kim and J. Jooyoung Lee, "A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs," *IEEE Electron Device Letters*, vol. 30, no. 8, pp. 846–848, aug 2009.

[23] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Invited - approximate computing with partially unreliable dynamic random access memory - approximate dram," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, pp. 100:1–100:4.

[24] I. Bhati, Z. Chishti, S.-L. Lu, B. Jacob, I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible auto-refresh," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 235–246, jun 2015.

[25] Z. Cui, S. A. McKee, Z. Zha, Y. Bao, and M. Chen, "DTail," in *Proceedings of the 28th ACM international conference on Supercomputing - ICS '14*. New York, New York, USA: ACM Press, 2014, pp. 43–52.

[26] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Efficient reliability management in SoCs - an approximate DRAM perspective," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, jan 2016, pp. 390–394.

[27] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, vol. 41, no. 3. New York, New York, USA: ACM Press, 2013, p. 48.

[28] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 5–5.

[29] S. Miucin, C. Brady, and A. Fedorova, "DINAMITE: A modern approach to memory performance profiling," *ArXiv e-prints*, Jun. 2016.

[30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[31] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275.

[32] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2007.

[33] Intel, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures," Sept. 2010.

[34] K.-L. Cheng, M.-F. Tsai, and C.-W. Wu, "Neighborhood pattern-sensitive fault testing and diagnostics for random-access memories," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1328–1336, Nov 2002.